REMARKS/ARGUMENTS

This Amendment is in response to the First Office Action mailed September 3, 2003. Claims 1-58 are now in this application. No claims have been canceled, and no claims have been added. Claims 55 and 56 are amended herein.

Claim Rejections – 35 U.S.C. § 102

Claims 1-4, 6-7, 9-13, 15, 17-23, 25-34, 38-41, 43-44, 46-50, 52 and 54-58 have been rejected under 35 U.S.C. § 102(b) as being anticipated by "Static Correlated Branch Prediction," Cliff Young and Michael D. Smith (hereinafter "Young"). The Applicant respectfully submits the following arguments pointing out significant differences between claims 1-4, 6-7, 9-13, 15, 17-23, 25-34, 38-41, 43-44, 46-50, 52 and 54-58 submitted by the Applicant and Young.

Young, as indicated by the title, is directed to methods for making branch predictions. (See also p. 1029, paragraphs 2-4) More specifically, Young discloses "a static technique for capturing both logical and statistical correlations." (See p. 1030, par. 2, line 1) Young defines a static technique as a software based prediction made by the compiler. (See p. 1029, par. 1, line 7 and par. 2, lines 7-9) Correlation is defined by Young as the understanding that "the outcome of a conditional branch is often determined by the branch's historical pattern of outcomes or the historical pattern of outcomes of its neighboring branches." (See p. 1029, par. 3, lines 1-4) In order "to identify all kinds of correlation during compilation, "[Young] uses[s] profile information to summarize a program's run-time behavior." (See p. 1030, par. 2, lines 2-3) Young then goes on to describe paths and path profiling beginning on page 1032 including an "algorithm for collecting path profiles of execution frequencies" beginning on page 1036. As can be seen in the figures and description of Section 2 of Young, a path profile is a representation of the possible paths control may take during execution of the program being compiled.

Additionally, Young describes, beginning on page 1040, an algorithm for making a static correlated branch prediction based on the path profile. This algorithm includes path profiling and local minimization as discussed beginning at the bottom of page 1040. Under Young, local minimization includes building a "history tree." (See p. 1042, par. 3 and 4) As described beginning on page 1044 at paragraph 4, a path profile is transformed into a history tree by the compiler running a program such as the one detailed in Figure 13. This program accumulates a

counter for the number of times a particular path through the history tree is traversed. (See p. 1043, par. 5)

In summary, Young describes branch predictions made by a compiler during compile time optimization. Under Young branch predictions can be made based on a history tree of possible paths. The history tree includes a frequency count for each node of the history tree. The frequency count is generated by the compiler during optimization based on the number of times a particular path is traversed. However, Young does not teach or suggest using a history operator and a history operand as described in the pending application.

As defined on page 10 of the description of the pending application, a history operand in source code represents a sequence of data associated with the history of an operand instance. Further, a history operator in source code represents a function that object code will perform on the history data represented by the history operand. Examples of history operators and history operands are shown in Figs. 4-12 of the pending application.

Independent claim 1, upon which claims 2-4, 6-7, 9-13, 15, and 17-18 depend, recites in part "recognizing a history operator and a history operand in the source code; generating first object code that, when executed, saves a data history associated with an instance of the history operand; and generating second object code that, when executed, performs the history operator on the data history." Young does not teach or suggest recognizing a history operator and a history operand in the source code, generating first object code that, when executed, saves a data history associated with an instance of the history operand, and generating second object code that, when executed, performs the history operator on the data history. Rather, Young teaches making a branch prediction based on a history tree of possible paths, the history tree including a frequency count for each node of the history tree generated by the compiler during optimization based on the number of times a particular path is traversed. For at least these reasons, independent claim 1 and its dependent claims 2-4, 6-7, 9-13, 15, and 17-18 are distinguishable from Young and should be allowed.

Independent claim 19, upon which claims 20-23 and 25 depend, recites in part "a history operand to direct a translator to generate first object code that, when executed, saves a data history associated with an instance of the history operand; and a history operator to direct the translator to generate object second code that, when executed, performs the history operator on the data history." Young does not teach or suggest a history operand to direct a translator to

13

generate first object code that, when executed, saves a data history associated with an instance of the history operand, and a history operator to direct the translator to generate object second code that, when executed, performs the history operator on the data history. Rather, Young teaches making a branch prediction based on a history tree of possible paths, the history tree including a frequency count for each node of the history tree generated by the compiler during optimization based on the number of times a particular path is traversed. For at least these reasons, independent claim 19 and its dependent claims 20-23 and 25 are distinguishable from Young and should be allowed.

Independent claim 26, upon which claims 27-30 depend, recites in part "recognizing a history operand in source code; finding at least one instance of the history operand in the source code in response to recognizing the history operand; allocating storage; and generating first object code associated with each instance, wherein the first object code, when executed, saves a data history associated with the history operand in the storage." Young does not teach or suggest recognizing a history operand in source code, finding at least one instance of the history operand in the source code in response to recognizing the history operand, allocating storage, and generating first object code associated with each instance, wherein the first object code, when executed, saves a data history associated with the history operand in the storage. Rather, Young teaches making a branch prediction based on a history tree of possible paths, the history tree including a frequency count for each node of the history tree generated by the compiler during optimization based on the number of times a particular path is traversed. For at least these reasons, independent claim 26 and its dependent claims 27-30 are distinguishable from Young and should be allowed.

Independent claim 31, upon which claims 32-34 depend, recites in part "recognizing a history operand in the source code, wherein the source code is contained in the memory; in response to recognizing the history operand, finding at least one instance of the history operand in the source code; allocating storage for a data history associated with the history operand; generating first object code associated with each instance, wherein the first object code, when executed, saves the data history associated with the history operand in the storage; and generating second object code that, when executed, performs the history operator on the data history." Young does not teach or suggest recognizing a history operand in the source code, wherein the source code is contained in the memory, in response to recognizing the history

operand, finding at least one instance of the history operand in the source code, allocating storage for a data history associated with the history operand, generating first object code associated with each instance, wherein the first object code, when executed, saves the data history associated with the history operand in the storage, and generating second object code that, when executed, performs the history operator on the data history. Rather, Young teaches making a branch prediction based on a history tree of possible paths, the history tree including a frequency count for each node of the history tree generated by the compiler during optimization based on the number of times a particular path is traversed. For at least these reasons, independent claim 31 and its dependent claims 32-34 are distinguishable from Young and should be allowed.

Independent claim 38, upon which claims 39-41, 43-44, 46-50, 52, and 54 depend, recites in part "recognizing a history operator and a history operand in the source code; saving a data history associated with an instance of the history operand; and performing the history operator on the data history." Young does not teach or suggest recognizing a history operator and a history operand in the source code, saving a data history associated with an instance of the history operand, and performing the history operator on the data history. Rather, Young teaches making a branch prediction based on a history tree of possible paths, the history tree including a frequency count for each node of the history tree generated by the compiler during optimization based on the number of times a particular path is traversed. For at least these reasons, independent claim 38 and its dependent claims 39-41, 43-44, 46-50, 52, and 54 are distinguishable from Young and should be allowed.

Independent claim 55, upon which claims 56-58 depend, recites in part "recognizing a history operand in source code, the history operand representing a sequence of data associated with the history of an operand instance; finding at least one instance of the history operand in the source code in response to recognizing the history operand; and saving a data history associated with each instance of the history operand in the storage." Further, claim 56 recites in part "recognizing a history operator in the source code, the history operator representing a function that object code will perform on the data history associated with the history operand; and performing the history operator on the data history." Young does not teach or suggest recognizing a history operand in source code, the history operand representing a sequence of data associated with the history of an operand instance, finding at least one instance of the history

operand in the source code in response to recognizing the history operand, and saving a data history associated with each instance of the history operand in the storage. Neither does Young teach or suggest recognizing a history operator in the source code, the history operator representing a function that object code will perform on the data history associated with the history operand and performing the history operator on the data history. Rather, Young teaches making a branch prediction based on a history tree of possible paths, the history tree including a frequency count for each node of the history tree generated by the compiler during optimization based on the number of times a particular path is traversed. For at least these reasons, independent claim 55 and its dependent claims 56-58 are distinguishable from Young and should be allowed.

Claims 35-37 have been rejected under 35 U.S.C. § 102(e) as being anticipated by Levine (USPN 6,349,406). Levine relates to "a method and system for compensating for instrumentation overhead in trace data by computing average minimum event times." (Abstract) More specifically, under Levine "in order to profile a program, the program is executed to generate trace records that are written to a trace file." (Col. 3, lines 16-18) The "trace data may be generated via selected events and timers through the instrumented interpreter without modifying the source code." (Col. 8, lines 14-16) The interpreter executes a trace program "used to record data upon the execution of a hook, which is a specialized piece of code at a specific location in a routine or program in which other routines may be connected." (Col. 9, lines 43-46) However, Levine does not teach or suggest using a history operator and a history operand as described in the pending application.

Independent claim 35, upon which claims 36 and 37 depend, recites in part "a first data field containing data representing a value associated with an instance of a history operand; and a second data field containing data representing a location within a program where the value was assigned." Levine does not teach or suggest a first data field containing data representing a value associated with an instance of a history operand; and a second data field containing data representing a location within a program where the value was assigned. Rather, Levine teaches trace data generated by an interpreter upon execution of a hook code in the source program. For at least this reason, claims 35-37 are distinguishable from Levine and should be allowed.

Claim Rejections – 35 U.S.C. § 103

Claims 5, 8, 14, 16, 24, 42, 45, 51 and 53 have been rejected under 35 U.S.C. § 103(a) as being unpatentable over Young, in view of Levine. For the reasons stated above, the combination of Young and Levine is no more relevant to the pending claims than either reference taken individually since neither reference teaches or suggests using a history operator and a history operand as described in the pending application.

Independent claim 1, upon which claims 5, 8, 14, and 16 depend, recites in part "recognizing a history operator and a history operand in the source code; generating first object code that, when executed, saves a data history associated with an instance of the history operand; and generating second object code that, when executed, performs the history operator on the data history." Neither Young nor Levine teaches or suggests recognizing a history operator and a history operand in the source code, generating first object code that, when executed, saves a data history associated with an instance of the history operand, and generating second object code that, when executed, performs the history operator on the data history. For at least these reasons, independent claim 1 and its dependent claims 5, 8, 14, and 16 are distinguishable from the combination of Young and Levine and should be allowed.

Independent claim 19, upon which claim 24 depends, recites in part "a history operand to direct a translator to generate first object code that, when executed, saves a data history associated with an instance of the history operand; and a history operator to direct the translator to generate object second code that, when executed, performs the history operator on the data history." Neither Young nor Levine teaches or suggests a history operand to direct a translator to generate first object code that, when executed, saves a data history associated with an instance of the history operand, and a history operator to direct the translator to generate object second code that, when executed, performs the history operator on the data history. For at least these reasons, independent claim 19 and its dependent claim 24 are distinguishable from the combination of Young and Levine and should be allowed.

Independent claim 38, upon which claims 42, 45, 51, and 53 depend, recites in part "recognizing a history operator and a history operand in the source code; saving a data history associated with an instance of the history operand; and performing the history operator on the data history." Neither Young nor Levine teaches or suggests recognizing a history operator and a history operand in the source code, saving a data history associated with an instance of the

17

history operand, and performing the history operator on the data history. For at least these reasons, independent claim 38 and its dependent claims 42, 45, 51, and 53 are distinguishable from the combination of Young and Levine and should be allowed.
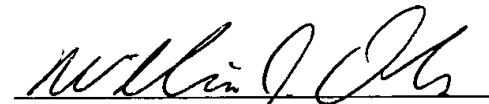
As all claims now in the application are in condition for allowance, Applicants request the application be allowed and pass to issuance as soon as possible.

It is believed that no further fees are due with this Response. However, the Commissioner is hereby authorized to charge any deficiencies or credit any overpayment with respect to this patent application to deposit account number 13-2725.

Respectfully submitted,

Dated: December _2_, 2003

27488
PATENT TRADEMARK OFFICE

William J. Daley, Reg. No. 52,471
Merchant & Gould P.C.
PO Box 2903
Minneapolis, MN 55402-0903
303.357.1651